Start here
Logistics

- Hwk1 here

- PA1: Any questions?
  - I posted a note on the news page about the solaris man page issue.
  - `man -3c wait`

- Hwk2 will be back very soon, Hwk3 by next Thursday.
Let’s pick up where we left off on multilevel scheduling.

Up to this point we looked at single queue scheduling
- FCFS
- SJF
- Priorities

Multilevel queues address the problem of scheduling when you have distinct classes of operations.
- E.g.: Foreground vs background

I muddled the distinction between multilevel and multilevel feedback scheduling, so let’s try to fix that.
Multilevel Queue

- Ready queue is partitioned into separate queues:
  foreground (interactive)
  background (batch)
- Each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS

Processes stay in their queue. Forever. Later we’ll see feedback-based methods that allow migration.
Multilevel Queue Scheduling

highest priority

- system processes

interactive processes

interactive editing processes

batch processes

student processes

lowest priority
Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way.
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service
Example of Multilevel Feedback Queue

- Three queues:
  - $Q_0$ – RR with time quantum 8 milliseconds
  - $Q_1$ – RR time quantum 16 milliseconds
  - $Q_2$ – FCFS
Multilevel Feedback Queues

- quantum = 8
- quantum = 16
- FCFS
Example of Multilevel Feedback Queue

- Scheduling
  - A new job enters queue $Q_0$ which is served RR. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue $Q_1$.
  - At $Q_1$ job is again served RR and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue $Q_2$.
  - $Q_2$ represents long running jobs, and are serviced only when $Q_1$ and $Q_0$ are finished. $Q_2$ lets long running jobs soak up cycles leftover by $Q_0$ and $Q_1$.
    - Jobs in $Q_1$ and $Q_2$ age as usual, and can percolate up in priority if they stagnate.

- Priority: $Q_0$ highest, $Q_1$ medium, $Q_2$ low.
MFQS Parameters

- Number of queues
- Scheduling algorithm per queue
- Method to determine when a process is upgraded to a higher priority queue.
- Method to determine when a process is downgraded to a lower priority queue.
- Method to determine which queue a process lands in when it needs service.
Thread Scheduling

- Distinction between user-level and kernel-level threads
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope (PCS)** since scheduling competition is within the process
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system
Process vs System Contention

- Process contention is where threads compete and the user thread library plays a role in deciding who gets priority.
- System contention is where threads compete with all threads in the system.

Threads within process contend with each other to get placed in system queues.
Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD SCOPE PROCESS schedules threads using PCS scheduling
  - PTHREAD SCOPE SYSTEM schedules threads using SCS scheduling.

- System scope means each pthread is a kernel thread, so it naturally competes with others.
  - One-to-one mapping of user to kernel

- Process scope means many-to-one mapping possible, with thread library deciding which thread resides on set of kernel threads at any time.
```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
    int i;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* set the scheduling algorithm to PROCESS or SYSTEM */
    pthread_attr_setscope(&attr, PTHREAD SCOPE SYSTEM);
    /* set the scheduling policy - FIFO, RT, or OTHER */
    pthread_attr_setschedpolicy(&attr, SCHED OTHER);
    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], &attr, runner, NULL);

    /* … */
```
Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
- **Processor affinity** – process has affinity for processor on which it is currently running
  - soft affinity
  - hard affinity
NUMA and CPU Scheduling

CPU

fast access

memory

computer

CPU

fast access

memory
Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consume less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens
Multithreaded Multicore System

- **C**: compute cycle
- **M**: memory stall cycle

Time sequence: C M C M C M C M
Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling
Solaris

- Six classes:
  - Time sharing
  - Interactive
  - Real time
  - System
  - Fair share
  - Fixed priority

- Default is time sharing.
  - Next slide we see the dispatch table for time sharing.
## Solaris Dispatch Table

<table>
<thead>
<tr>
<th>priority</th>
<th>time quantum</th>
<th>time quantum expired</th>
<th>return from sleep</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>200</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>160</td>
<td>0</td>
<td>51</td>
</tr>
<tr>
<td>15</td>
<td>160</td>
<td>5</td>
<td>51</td>
</tr>
<tr>
<td>20</td>
<td>120</td>
<td>10</td>
<td>52</td>
</tr>
<tr>
<td>25</td>
<td>120</td>
<td>15</td>
<td>52</td>
</tr>
<tr>
<td>30</td>
<td>80</td>
<td>20</td>
<td>53</td>
</tr>
<tr>
<td>35</td>
<td>80</td>
<td>25</td>
<td>54</td>
</tr>
<tr>
<td>40</td>
<td>40</td>
<td>30</td>
<td>55</td>
</tr>
<tr>
<td>45</td>
<td>40</td>
<td>35</td>
<td>56</td>
</tr>
<tr>
<td>50</td>
<td>40</td>
<td>40</td>
<td>58</td>
</tr>
<tr>
<td>55</td>
<td>40</td>
<td>45</td>
<td>58</td>
</tr>
<tr>
<td>59</td>
<td>20</td>
<td>49</td>
<td>59</td>
</tr>
</tbody>
</table>

- Low priority = 0

- Time quantum increases as priority decreases.

- When threads use entire time quantum without blocking, they are demoted.

- Threads that return from waiting (e.g.: IO) are boosted to higher priorities.
Solaris Scheduling

Scheduling priorities between classes

- Interrupt threads
- Realtime (RT) threads
- System (SYS) threads
- Fair share (FSS) threads
- Fixed priority (FX) threads
- Timeshare (TS) threads
- Interactive (IA) threads
## Windows XP Priorities

### Priority classes

<table>
<thead>
<tr>
<th>Relative priorities within a class</th>
<th>real-time</th>
<th>high</th>
<th>above normal</th>
<th>normal</th>
<th>below normal</th>
<th>idle priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>time-critical</td>
<td>31</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>highest</td>
<td>26</td>
<td>15</td>
<td>12</td>
<td>10</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>above normal</td>
<td>25</td>
<td>14</td>
<td>11</td>
<td>9</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>normal</td>
<td>24</td>
<td>13</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>below normal</td>
<td>23</td>
<td>12</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>lowest</td>
<td>22</td>
<td>11</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>idle</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Linux Scheduling

- Constant order $O(1)$ scheduling time
- Two priority ranges: time-sharing and real-time
- **Real-time** range from 0 to 99 and **nice** value from 100 to 140
- (figure 5.15)
O(1) Scheduling

- What does this O(1) mean? It was introduced in the 2.6.x family of kernels.

- Previously the scheduler was O(N), where N was the number of tasks in the system.
  - More tasks, more overhead to schedule a task.

- Now scheduling is based on finding the first process in a fixed set of queues (one queue per priority level).
  - NOT a function of task count. No matter how many tasks, same overhead to schedule a task.

- O(1) should be read as “constant time”.
  - NOT unit (1) time.
### Priorities and Time-slice length

<table>
<thead>
<tr>
<th>numeric priority</th>
<th>relative priority</th>
<th>time quantum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>highest</td>
<td>200 ms</td>
</tr>
<tr>
<td>99</td>
<td></td>
<td>real-time tasks</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>other tasks</td>
</tr>
<tr>
<td>140</td>
<td>lowest</td>
<td>10 ms</td>
</tr>
</tbody>
</table>
List of Tasks Indexed According to Priorities

active array

priority: [0] [1] [140]
task lists: • • •

expired array

priority: [0] [1] [140]
task lists: • • •
Algorithm Evaluation

- Deterministic modeling – takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Queueing models
  - Simulation is composed of a set of queue data structures.
  - Tasks are moved between queues.
- Implementation

- One can also use stochastic models
  - Instead of a workload of actual times, measure the probability distribution of a real workload and create surrogate workloads by sampling distributions.
  - These are actually not too hard to implement.
Evaluation of CPU schedulers by Simulation

- Simulation for FCFS
- Simulation for SJF
- Simulation for RR (q = 14)

Actual process execution:
- CPU 10
- I/O 213
- CPU 12
- I/O 112
- CPU 2
- I/O 147
- CPU 173

Trace tape
PA2 Idea

I was thinking about PA2, and had an idea.

Current intent: You will modify the Linux kernel to add a system call.
- Not complex w.r.t. the system call, but complexity is in adding code, figuring out how to build, etc…

Other possibility:
- Implement a simple queuing network model, and get a hands on feel for how scheduling algorithms work by implementing them in the QNM.

Thoughts? There is a tradeoff:
- Kernel modification = hands dirty in a real OS
- QNM = more complex logic, but you can actually play with a scheduler.
End of Chapter 5